Gate-Level Synthesis and Technology Mapping

Christian Krieg

November 12, 2024

Welcome to this year's *Digital Integrated Circuits* course! Throughout this course, we will have to solve some assignments that will teach us fundamental concepts of integrated circuit design.

We will demonstrate digital integrated circuit design on a step-by-step simple example of a digital counter. As the name of this course suggests, our ultimate goal is to implement this counter as a digital integrated circuit. In the lab, we map this counter to a field-programmable gate array (FPGA) architecture, and run it on the board shown in Figure 1. The counter's functionality is executed on the FPGA. We will be able to control execution by pushing the buttons, and to visualize the counter value using the light-emitting diodes (LEDs). Because operating a hardware counter with buttons and LEDs may be inconvenient, we also will interface the hardware counter over a universal asynchronous receiver/transmitter (UART), so that we may be able to further process the counter value on a host computer. But this will all happen during the lab week at the end of the exercises.

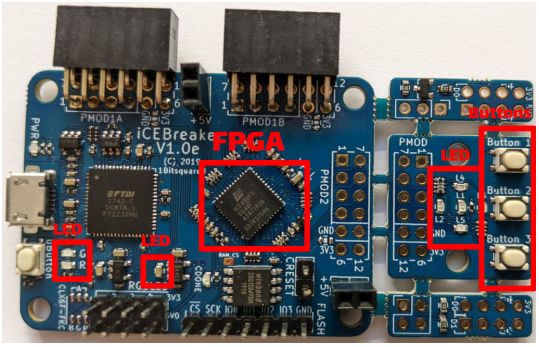

Figure 1: The *icebreaker* board, which will run our counter

Before we can bring our counter on hardware, it is important that we learn how to design digital circuits using state-of-the-art (and beyond) methods and tools, and to get a feeling for the results of our design decisions. We will draw attention on the results of the synthesis steps, which we will verify with several tools along the digital design flow.

Generally speaking, we perform the following steps:

1. Specify the counter's behavior
2. Model (implement) the counter in a hardware description language (HDL)
3. Verify the counter's implementation
4. Synthesize and optimize the counter's HDL model + verify
5. **Map the counter to the target (FPGA) technology + verify**
6. Generate an FPGA bitstream + verify
7. Configure the FPGA with the bitstream + verify
8. Run the counter on hardware

Our overall task is to build a synchronous counter that runs in two modes, based on a *mode* switch. The two modes are called *up* and *down*. In *up* mode, the counter increments the counter value by **5** at the rising edge of the clock signal, and in *down* mode, it decrements the counter value by **9** at the rising edge of the clock signal. The counter should implement synchronous reset behavior, such that it is set to an initial value of **-50** upon reset. The counter should never exceed a value of **235**, and should never go below a value of **-230** (the counter value sticks near the minimum/maximum value, until the counting direction changes). The counter value should never be **-11** (the counter should jump over this value by one increment/decrement).

Listing 1: Yosys commands to select the register of our optimized counter

```
# Read design and synthesize processes
read_verilog counter.v
proc
clean
opt -full

# Select all d-type registers
select t:$dff

# Print the selected part of the netlist
dump

# Clear selection (selects entire design again)
select -clear

# Print netlist of entire design
dump
```

Glossary

ASIC Application-specific integrated circuit. 7

CPLD Complex programmable logic device. 7

CSV Comma-separated values. 9

FPGA Field-programmable gate array. 1, 7

HDL Hardware description language. 1

LED Light-emitting diode. 1

LUT Lookup table. 8, 10

RTL Register transfer level. 2, 6–8

UART Universal asynchronous receiver/transmitter. 1

Gate-level synthesis and technology mapping

In this task, we will learn to use the *select* infrastructure provided by *Yosys*. The *select* command is very useful to investigate a design, and to apply operations only to a subset of a design. This is considerably useful with growing design complexity. As we already noticed, design complexity steadily increases with lower abstraction levels. We will demonstrate the power of *select* by gradually performing gate-level synthesis on our optimized register transfer level (RTL) design.

Design selections

Have a look at the *select* command's help text. You will find an exhaustive description of the select syntax, which may be unintuitive at the first glimpse. However, the select syntax describes a powerful, sophisticated method to partially select a design and perform operations on it.

Our first task will be to search for our counter's register. In the following, we perform all operations on our optimized counter (*counter_opt.v*). We know that our counter only has one register, and that this register is a D-type flipflop (have a look at the *stat* command's output). So, to select our counter register, we select all cells of type *$dff* (Listing 1).

In Listing 1, we select the only register in the design, and dump its instantiation. If we had more registers in the design, information on each register would be printed. The dumped output should look like what is given in Listing 2. The *$dff* cell has a *src* attribute which specifies the location in the original source from which the cell is inferred (Line 1). In the example given in Listing 2, the *$dff* is inferred from the code given in file *counter_opt.v* in line 795. Line 2 specifies the dumped object (it's a *cell*!), its type (*$dff*), and its name (*$procdff$269*). Lines 3 and 4 define the cell's parameters: the register is sensitive to the positive clock edge (`\CLK_POLARITY 1'1`), and the register is ten bits wide (`\WIDTH`

Listing 2: Dump of the counter's register

```
1  attribute \src "counter.v:795"
2  cell $dff $procdff$269
3    parameter \CLK_POLARITY 1'1
4    parameter \WIDTH 10
5    connect \CLK \clk
6    connect \D \_262_
7    connect \Q \cnt
8  end
```

Listing 3: Different ways to work with selections

```
1 # First way: Select and perform command
2 select t:$dff
3 show
4 stat
5 select -clear
6
7 # Second way: Perform command with selection as last argument
8 show t:$dff
9 stat t:$dff
```

10). Lines 5 to 7 define the cell's connectivity: the register's \CLK port is connected to a net named \clk, the register's data input \D is connected to net named _262_, and the register's output is connected to a net named \cnt.

You may have already noticed that the majority of *Yosys* commands operates on selections. Up to now, we used all Yosys commands on the entire design (like *show*, *stat*, *techmap*, etc.). However, you can also use all these commands on a subset of the design. Just try it out. Listing 3 shows two ways of working with selections: the first way is to select the design and then perform the command; the second way is to perform the command, giving the selection as last argument.

One single cell is not very much to investigate. So we will also select what is connected to the flip-flop's inputs by using the *input cone* selection feature. Listing 4 shows how it works. But let us now use the register's name explicitly, instead of selecting all cells of type *$dff*. Because the register is part of the module *counter*, we have to specify the entire path (*counter/$procdff$269*). We select one level of the input cone by adding *%ci*. Have a look at the selection's dump output. Besides the register, we also selected the wires that are connected to the register's inputs. This becomes also visible in the *show* command's output: The non-selected wires visualized as ellipses turn to diamonds (for internal wires) and polygons (for primary inputs/outputs).

There are two variants of selecting the input cone: *%ci*, and *%cie*. The difference between these two options is that *%ci* considers every cell, whereas *%cie* only considers combinational cells. This is useful for selecting the combinational parts between registers. We can gradually increase the input cone by adding the number of levels to *%ci* and *%cie*. For the entire input cone, we add an asterisk (*) to *%ci* and *%cie*. The input cone consists of cells and wires, which are alternating in the respective levels. Have a look at the effects by executing the commands given in Listing 5.

We realize that we select the entire module when we execute the command given in Line 9. This is because it only consists of one register, and we select the entire combinational logic connected to this register (implementing the counter's behavior).

Now, let's have a look at the output of the command given in Line 6. We notice that we selected a multiplexer that is controlled by the *mode* input. As we know from the system specification, the mode signal switches between counting up (*mode* = 1) and counting down (*mode* = 0). Let's say we only want to select the logic that implements the increment. To do so, we have to select the logic that is connected to the port of the multiplexer that propagates the increment's result to its output, which in our case is port \B. We can do so by first selecting the multiplexer that is controlled by the *mode* signal (let's have a look at all multiplexers in the design as shown in Listing 6). Then visualize it using the 'show' command. Note that the multiplexer name given in Line 5 is just an example. It will be different in your design.

Have a look at Listing 7 to figure out how we select everything that is connected to this multiplexer's \B input. We can do so by selecting only the wire that is connected to the multiplexer's \B input (Line 7), and then select the input cone

Listing 4: Select one input cone level

```
1 select counter/$procdff$269 %ci
2 dump
3 show
```

Listing 5: Select input cone of gradually increasing depth

```
show counter/$procdff$269 %ci
show counter/$procdff$269 %ci %cie
show counter/$procdff$269 %ci %cie2
show counter/$procdff$269 %ci %cie3
show counter/$procdff$269 %ci %cie4
show counter/$procdff$269 %ci %cie5
show counter/$procdff$269 %ci %cie6
show counter/$procdff$269 %ci %cie7
show counter/$procdff$269 %ci %cie*
```

Listing 6: Show the multiplexer that is controlled by the *mode* switch

```
# Dump all multiplexers
dump t:$mux

# Select the multiplexer that is controlled by the 'mode' input
show $ternary$counter.v:800$261
```

to this wire (Line 10).

However, the command given in Listing 7 is somewhat hard to read. So, let's give a more readable command that better describes what we aim to do in Listing 8 using *named selections* and output cone (*%co*). As given in Line 2, we select the wire with the name *mode*, and select one level of output cone to capture the multiplexer connected to it. We remove the *mode* wire from the selection again to only select the multiplexer. A selection can be named and stored for later use by using the *set* option. When used, the named selection is de-referenced with a prefix '@'. We select and store the counter's increment network in Line 5, and its decrement network in Line 8. We use the named selections in Lines 11, 12, 15 and 16.

Extracting submodules

Yosys provides another very useful feature for design analysis: submodule extraction. Let's say we have selected our increment and decrement networks, and we now would like to further analyze them. For example, we would like to simulate them separately, or we would like to create multiple instances in order to apply different optimization strategies. Submodule extraction is implemented in the *submod* command. We extract our increment and decrement networks with the commands given in Listing 9.

Have a look what happens: Print the list of modules with *ls*. Instead of one module, *counter*, we now have three modules: *counter*, *increment*, and *decrement*.

We can quickly reverse this by using the *flatten* command, which replaces the module instances by their implementations. However, this is just for your info; **we don't do this yet**.

Gate-level synthesis

We are now at a point where we can synthesize our design to gate level. Note that it is not necessary at all to extract submodules for gate-level synthesis. The reason we do it here is to get a better understanding of what is going on in this synthesis step. With the entire counter, we quickly could get lost in design complexity.

So, let's get started. We synthesize the increment and decrement network to gate level using the *techmap* command as shown in Listing 10 (Lines 2 and 3). Use *stat* and *show* to see the results of gate-level synthesis (Lines 6, 7, 10 and 11). Multi-bit cells are mapped to single-bit cells with 2-inputs and one output. For this reason, also the cell count increases. However, we can still optimize the design (Line 14). Have a look again at the results using *stat* and *show*.

Listing 7: Selecting the logic network that connects to the multiplexer's 'B' input

```
# Select the multiplexer, including its 'B' input
show $ternary$counter.v:800$261 %ci:+[B]

# Select the multiplexer, including one level of input cone to its 'B' input,
# then remove the multiplexer to only select the wire(s) connecting to the 'B'
# input
show $ternary$counter.v:800$261 %ci:+[B] $ternary$counter.v:800$261 %d

# Select only the logic that connects to the multiplexer's 'B' input
show $ternary$counter.v:800$261 %ci:+[B] $ternary$counter.v:800$261 %d %cie*
```

Listing 8: Using named selections to select the increment/decrement networks

```
# Create a named selection for the multiplexer that is controlled by the 'mode' input
select -set modeswitch w:\mode %co w:\mode %d

# Select only the logic that connects to the multiplexer's 'B' input and name it 'inc'
select -set inc @modeswitch %ci:+[B] @modeswitch %d %cie*

# Select only the logic that connects to the multiplexer's 'A' input and name it 'dec'
select -set dec @modeswitch %ci:+[A] @modeswitch %d %cie*

# Print statistics and show increment network
stat @inc
show @inc

# Print statistics and show decrement network
stat @dec
show @dec
```

Listing 9: Extracting modules

```
# Create submodules
submod -name increment @inc
submod -name decrement @dec
clean

# Show the list of modules
ls

# Draw the counter's schematic
show counter

# Make everything undone
flatten
opt -full
```

Listing 10: Gate-level synthesis

```
# Gate-level synthesis
techmap increment
techmap decrement

# Watch the result for 'increment'
stat increment
show increment

# Watch the result for 'decrement'
stat decrement
show decrement

# Optimize the gate-level design
opt -full

# Watch the result again!
```

Listing 11: Write our modular design to Verilog files

```
# Write the 'increment' module
cd increment
write_verilog -selected increment.v

# Write the 'decrement' module
cd decrement
write_verilog -selected decrement.v

# Write the 'counter' module
cd counter
write_verilog -selected counter_submod.v
```

Listing 12: Verify our mixed RTL/gate-level counter

```
[options]
mode prove
expect pass
depth 100

[engines]
smtbmc --syn

[script]
read -formal properties.sv counter_submod.v increment.v decrement.v
verific -import ctb
prep -top ctb

[files]
properties.sv
counter_submod.v
increment.v
decrement.v
```

We can now write the result of the gate-level synthesis to Verilog files. We could write all the modules together into one Verilog file. However, as we created a modular design, we will write every module into a separate file, using the *-selected* option. Listing 11 shows how it is done.

Design verification

We successfully identified the increment/decrement networks of our counter, separated them into modules, synthesized them to gate level, and created a modular design in separate Verilog files. But hey! Does the counter still behave as specified? To answer this question, we can still use our properties to formally verify with *SymbiYosys* whether they hold on our counter! All we have to do is to adjust the *SymbiYosys* configuration, as given in Line 7. Please note the *--syn* option in Line 7. It is necessary because of the mixed RTL/gate-level design. Without this option, the solver thinks that it detects a logic loop – which in fact is an artifact due to the mixed representations.

Bringing everything to gate level

Finally, let's synthesize the entire counter to gate level. We first flatten the design in order to replace the *increment* and *decrement* instances by their gate-level implementations. In a second step, we synthesize the rest of the design (which basically are the register and the two multiplexers that implement the mode switch and the reset behavior). Have a look at Listing 13 to see how we do it. Note the different cell types when inspecting the result using *stat* and *show* (Lines 6 and 7). In particular, we clearly see two remaining cell types at RTL: One *$dff*, and two *$mux*. After performing gate-level synthesis (Line 10), these cells are gone, but replaced by their gate-level representations. We see that instead of the *$dff* cell, we now have 10 *$_DFF_P_* cells, one for each bit of our counter register. Have a look using *stat* and *show* again (Lines 13 and 14). We can again optimize our design (Line 17), but it could be the case that the optimizer does not find anything that could be optimized any more. Finally, let's write our gate-level counter to a Verilog file (Line 24).

Map to target technology

We already applied a lot of synthesis steps to our original specification. We implemented a behavioral model of our counter, we replaced the abstract *addition* operation by concrete structural implementation, we optimized the design, and we synthesized a gate-level netlist from it.

However, our ultimate goal is to bring our counter on hardware. And yet we do not have an idea how all these design and synthesis steps influence the final outcome of the synthesis process. This is what we investigate in the following:

Listing 13: Synthesizing the entire counter to gate level

```
# Flatten the design
flatten
opt -full

# Watch the result
stat
show

# Gate-level synthesis
techmap

# Watch the result again
stat
show

# Optimize design
opt -full

# Watch the result again
stat
show

# Write the counter to file
write_verilog counter_gl.v
```

We will create technology-mapped netlists for any design representation we created so far, and we will investigate the resources that are required by each of the implementation.

In order to manage all the different synthesis results without having to write every single representation to its own Verilog file, we learn that Yosys can manage multiple designs in one session. We will learn the essential commands for saving and loading designs, and to perform operations on that designs.

We finally learn to load user-defined functionality into Yosys using its plugin infrastructure. The plugin infrastructure reveals the true power of Yosys's open-source nature: the freedom to extend it's functionality by individual needs.

Technology mapping

The next logical step in our synthesis and verification flow is to map our design to target hardware. In principle, this could be any target, including full-custom application-specific integrated circuits (ASICs), complex programmable logic devices (CPLDs), and FPGAs. Because ASIC design and verification widely goes beyond the scope of this course, we will implement our design on reconfigurable logic, in particular, on the Lattice iCE40 FPGA architecture. You will find details on the iCE40 architecture in the corresponding data sheet[1]. Our goal in this step of the synthesis flow is to map our netlist on the available logic in the target technology. The available logic cells in the iCE40 technology are documented in the Lattice iCE Technology Library[2].

Are you interested where you find the technology library for the Yosys toolchain? Just have a look at the *techlibs*[3] directory, where you find resources for all supported architectures, including iCE40 (*ice40* subdirectory). Just like our *map.v* from Task 3, these files hold implementations for the logic available on the hardware.

For verification purposes, the file *cells_sim.v* provides simulation models that model the behavior of the target technology primitives. We will need them to verify our counter's functionality with our properties.

Yosys provides a one-stop-shop command for synthesizing a design to the iCE40 target technology: *synth_ice40*. Just have a look at *help synth_ice40* to find out everything about that synthesis step. You will realize that a lot synthesis routines are applied to the design by *synth_ice40*.

So the question is now: What is the best representation of our design to provide to *synth_ice40*? Is it the fresh behavioral model of our counter? Or is it the version where we mapped our structural implementation of *$add*? If so, should we provide the RTL model, or should we provide the gate-level model?

As we do not have an answer to these questions yet, we will find out ourselves by mapping any of these representations to the iCE40 technology. We will then investigate the impact of each synthesis step on the consumed resources. As we do not want to write, read, and manage Verilog files for each and every one of these representations, we use the ability of Yosys to hold multiple designs in one session.

[1] http://www.latticesemi.com/ /media/LatticeSemi/Documents/DataSheets/iCE/iCE40LPHXFamilyDataSheet.pdf
[2] http://www.latticesemi.com/ /media/LatticeSemi/Documents/TechnicalBriefs/SBTICETechnologyLibrary201608.pdf
[3] https://github.com/YosysHQ/yosys/tree/master/techlibs

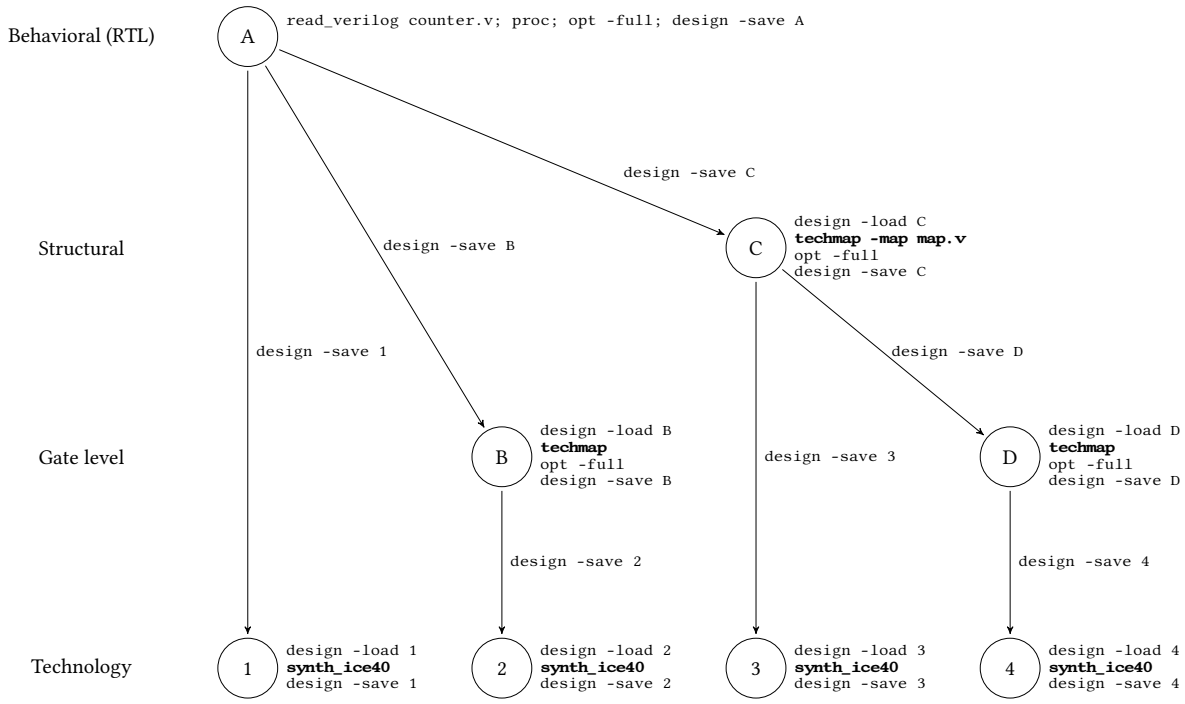

Figure 2: Visualization of creating a technology-mapped netlist from all design representations we have created so far.

Design handling

Yosys supports a very useful feature: managing multiple designs in one session, having any design present in memory, without the need of writing and reading files. Design handling is implemented by the *design* command. A design can be saved with *design -save <name>*, and it can be loaded as the *current design* using *design -load <name>*. That's it!

We now would like to create a technology-mapped netlist from any representation of our counter we created so far. In order to do so systematically, we create a map that organizes all the necessary steps as given in Figure 2.

What we basically see in Figure 2 is how we can get four technology-mapped netlists from our behavioral counter model, by varying the design and synthesis steps we have learned so far. Each node represents a design, for which we create a design object in Yosys. The nodes with numbers represent four technology-mapped netlists we want to generate; the nodes with letters are intermediate designs we need to create for generating the technology-mapped netlists. Node *A* represents the behavioral model of our counter. We generate this representation by the sequence of Yosys commands that is given right next to the node. Every other design representation is derived from the behavioral model. Each design representation can be generated by following the path, starting at node *A*. For example, to generate the technology-mapped netlist for the behavioral model, we do nothing but calling *synth_ice40*, thus creating the design that is represented by node *1*. If we want to create the technology-mapped netlist for the RTL model of the counter that uses the structural *$add* implementation, we follow the path from *A* to *3*, which is $A \rightarrow C \rightarrow 3$, executing every command given for this path. The exact command sequence for this particular example is given in Listing 14. Note that all these *design -save* and *design -load* operations are only necessary because we want to generate four techmapped netlists in a very systematic way. If we only would like to generate one techmapped netlist, we are very fine to just give the command sequence without the *design* commands.

Let's have a look at the resources that are consumed by the technology-mapped netlist of the counter that we created in Listing 14. *stat* will tell us something very similar as given in Listing 15. We notice that the counter now is implemented with cells from the iCE40 technology library, comprising flipflops (*SB_DFFE*, *SB_DFFESR*), 4-input lookup tables (LUTs) (*SB_LUT4*), and carry-logic cells (*SB_CARRY*). This is very interesting, because it shows us that some of the arithmetic logic is mapped to very efficient carry logic provided by the target technology. Do you still remember Task 4? Here, we explicitly implemented carry logic for the adder cells.

Our goal is to obtain the *stat* output for all four technology-mapped netlist variations. Therefore, your task is to write a Yosys script *techmap_synth.ys*, that creates all four technology-mapped netlists like in Listing 14, based on Figure 2.

Listing 14: Yosys command sequence to map the RTL model of our counter to the iCE40 target architecture, for which the *$add* cells were mapped to structural implementations (snippet of *techmap_synth.ys*)

```
1  read_verilog counter.v
2  proc;
3  opt -full;
4  design -save 1
5  design -save C
6
7  design -load C
8  techmap -map map.v; opt -full
9  design -save C
10 design -save 3
11
12 design -load 3
13 synth_ice40
14 write_verilog -noexpr 3.v
15 design -save 3
```

Listing 15: Output of *stat* for the technology-mapped netlist created in Listing 14

```
1    Number of wires: 94
2    Number of wire bits: 153
3    Number of public wires: 4
4    Number of public wire bits: 13
5    Number of memories: 0
6    Number of memory bits: 0
7    Number of processes: 0
8    Number of cells: 136
9      SB_CARRY 46
10     SB_DFFE 4
11     SB_DFFESR 6
12     SB_LUT4 80
```

Verification

We can still use our properties to verify that our counter behaves as specified. Just write a Verilog file for each technology-mapped netlist. Choose as file name the number of the design (e.g., *3.v* for the netlist created in Listing 14, Line 14). Note the *-noexpr* option; it will instruct Yosys to write a structural netlist rather than a behavioral Verilog model. We can then write a *SymbiYosys* configuration as given in Listing 16. We define four tasks, one for each techmapped netlist. When calling with *sby -f check.sby*, *SymbiYosys* checks all four representations of the counter. Note Line 26 in Listing 16. Here we tell SymbiYosys where it can find the simulation models for the iCE40 technology cell library. The simulation models are actually read in Line 16.

Plugin infrastructure

Remember, our goal is to compare resource allocations for technology-mapped netlists that have been created from different representations of our counter. In order to do so, we need to analyze the output of *stat* for each representation. To get an overview, it is a good idea to arrange the output in a table, one line for each techmapped netlist, and to arrange the allocated resources in the columns.

Because it is very error-prone to copy-paste the output for every *stat* output into a table, we wrote a Yosys plugin to automatically create such a table. The name of this plugin is *stat_csv*, and it creates a comma-separated values (CSV) file with the desired output. Simply load and call the plugin as given in Listing 17. *stat_csv* generates a file *counter.csv*. You can watch it using a text editor, but probably you will get a better overview if you use a spreadsheet editor such as *LibreOffice Calc*.

Comparing implementations

Table 1 shows an example table created by *stat_csv*. Your table will look very similar. What we can read from the table is quite interesting. We see that for design 1, the most carry logic is instantiated. This is because the behavioral model is directly synthesized into a techmapped netlist, and therefore the mapping algorithm has the highest degree of freedom in mapping to the target technology. Have a look at the number of instantiated carry cells for design 3. The number of instantiated carry logic decreases, because we gradually define the implementations of the arithmetic cell (*$add*). However, the number of LUTs increases, and also the number of wires. This is important for the following synthesis steps, which include placement and routing. Wires can have significant impact on the circuit's timing behavior. We also notice that the techmapped netlists, which are synthesized from the gate-level representations (designs 2 and 4), have significantly smaller cell and wire counts, and do not instantiate built-in carry logic. This can have significant impact on timing behavior too. We also notice that every design has the same number of public wires and public wire bits. The

Listing 16: *SymbiYosys* configuration to verify functional correctness of all four technology-mapped netlists (*check.sby*)

```
[tasks]
1
2
3
4

[options]
mode prove
expect pass

[engines]
smtbmc

[script]
read -define NO_ICE40_DEFAULT_ASSIGNMENTS
read -formal cells_sim.v
read -formal properties.sv
1: read -formal 1.v
2: read -formal 2.v
3: read -formal 3.v
4: read -formal 4.v
verific -import ctb
prep -top ctb

[files]
/usr/local/tabby/share/yosys/ice40/cells_sim.v
properties.sv
1: 1.v
2: 2.v
3: 3.v
4: 4.v
```

Listing 17: Commands to load and call *stat_csv*, plugin to create a csv file with output of the *stat* command (to be added to *techmap_synth.ys*)

```
plugin -i stat-csv.so
stat_csv
```

public wires connect the design to the "outside world", which in fact are the signals of the top-level entity: *cnt* (10 bits), *mode* (1 bit), *clk* (1 bit), and *rst* (1 bit).

Table 1: Example table created by *stat_csv*

Design	Wires	Bits	Public wires	Public bits	Total Cells,	SB_DFFSR	SB_LUT4	SB_CARRY	SB_DFF
1	34	169	4	13	98	2	45	43	8
2	38	153	4	13	61	2	51	0	8
3	45	190	4	13	100	2	56	34	8
4	36	124	4	13	56	2	46	0	8

Submission instructions

You are expected to reach the same resource utilisation as VELS. Unfortunately there seems to be a bug in Yosys[4] that makes the exact resource consumption dependent of the order of design save and load statements in your *techmap_synth.ys*.

The following order should work:

```
design -save A
design -save 1
design -save B
design -save C

design -load B
...

design -load C
...

design -load D
...

design -load 1
...

design -load 2
```

[4] https://github.com/YosysHQ/yosys/issues/2462

```
...

design -load 3
...

design -load 4
...
```

Attach the following files to your submission e-mail:

1. *counter.v*
2. *map.v*
3. *increment.v*
4. *decrement.v*
5. *counter_submod.v*
6. *counter_gl.v*
7. *techmap_synth.ys*